Christu Jyothi Institute of Technology & Science

JANGAON

COMPUTER SCIECNE ENGINEERING DEPARTMENT

B.TECH – III YEAR II SEMESTER

(Regulation: R22)

(2025-2026)

FULL STACK DEVELOPMENT LAB MANUAL

(CS611PE)

Prepared By: Lagishetti Suresh *M.Tech(CSE) – CSE, CJITS*

List of Experiments

1. Create an application to setup node JS environment and display “Hello World”.

2. Create a Node JS application for user login system.

3. Write a Node JS program to perform read, write and other operations on a file.

4. Write a Node JS program to read form data from query string and generate response using NodeJS

5. Create a food delivery website where users can order food from a particular restaurant listed in the website for handling http requests and responses using NodeJS.

6. Implement a program with basic commands on databases and collections using MongoDB.

7. Implement CRUD operations on the given dataset using MongoDB.

8. Perform Count, Limit, Sort, and Skip operations on the given collections using MongoDB.

9. Develop an angular JS form to apply CSS and Events.

10. Develop a Job Registration form and validate it using angular JS.

11. Write an angular JS application to access JSON file data of an employee from a server using $http service.

12. Develop a web application to manage student information using Express and Angular JS.

13. Write a program to create a simple calculator Application using React JS.

14. Write a program to create a voting application using React JS

15. Develop a leave management system for an organization where users can apply different types of leaves such as casual leave and medical leave. They also can view the available number of days using react application.

16. Build a music store application using react components and provide routing among the pages.

17. Create a react application for an online store which consist of registration, login, product information pages and implement routing to navigate through these pages.

Prepared By: Lagishetti Suresh *M.Tech(CSE) – CSE, CJITS*

EXPERIMENT – 1

1. Create an application to setup node JS environment and display "Hello World".

AIM: To set up the Node.js development environment and create a simple application that displays **"Hello World"** on the console and/or browser.

OBJECTIVE:

- To install and configure Node.js.
- To initialize a Node.js project.
- To write a basic JavaScript program using Node.js.
- To execute a Node program in the terminal.
- To create a simple HTTP server (optional).

THEORY:

Node.js is an open-source, cross-platform runtime environment built on Chrome's V8 JavaScript engine. It allows developers to execute JavaScript code outside the browser. Node.js is widely used for backend development, API creation, and real-time applications.

Key points:

- Uses asynchronous & event-driven architecture
- High performance due to V8 engine
- Supports large ecosystem (NPM modules)

PROCEDURE

Step 1: Install Node.js (first you need to install Python in the system).

Type the command at command prompt: **C:\> pip install nodejs-bin (After Install check below commands)**

C:\> node –v

C:> npm -v

Step 2: Create Project Folder

C:\> md exp1

C:\> cd exp1

Step 3: Initialize Node Project

npm init -y

Step 4: Create `app.js` File

Write the following code: (Notepad or visual Studio IDE)

```
const http = require("http");

const server = http.createServer((req, res) => {
   res.writeHead(200, { "Content-Type": "text/plain" });
   res.end("Hello World");
});

server.listen(3000, () => {
   console.log("Server running at http://localhost:3000");
});
```

Step 5: Run the Application (Goto the command prompt at same directory)

Type following command:

C:\exp1> node app.js

Server running at http://localhost:3000

(Copy given URL at Browser Address Bar for Output).

EXPERIMENT – 2

2. Create a Node JS application for user login system.

AIM:
To create a simple Node.js application that connects to MySQL to register a user (with hashed password) and allow login verification.

OBJECTIVE

- Connect Node.js app with MySQL database.
- Create `users` table.

THEORY (brief)

Node.js allows creating servers using the built-in `http` module. The **MySQL2** module enables Node.js to connect to a MySQL database.

PROCEDURE:

1. Create project folder and initialize:

```
C:\> md exp2
C:\>cd exp2
C:\exp2>npm init -y
```

Process:

1. Browser sends login form data (POST request).
2. Node.js server reads the request body manually using events (`data` and `end`).
3. The `querystring` module converts the POST data into key-value format.
4. A SQL query checks whether username & password exist in the MySQL `users` table.
5. The server responds with either *Login Successful* or *Invalid Credentials*.

This experiment shows how backend validation works without frameworks.

PREPARATION / SQL (run in MySQL):

```
CREATE DATABASE IF NOT EXISTS lab_auth;
USE lab_auth;

CREATE TABLE IF NOT EXISTS users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(100) NOT NULL UNIQUE,
  email VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

PROCEDURE

1. Install Node.js and MySQL on the system.
2. Create project folder and files:
   - `server.js`
   - `login.html`
   - `styles.css`
3. Install MySQL module:

   C:\> npm install mysql2

4. Start MySQL and create table using SQL above.
5. Write code in `server.js` to:

   - Connect to database
   - Serve HTML and CSS files
   - Receive POST request
   - Validate credentials

6. Run server:

   C:\exp2>node server.js

7. Open browser:

   http://localhost:3000

PROGRAM(CODE):

Server.js:

```
const http = require("http");
const fs = require("fs");
const querystring = require("querystring");
const mysql = require("mysql2");

// MySQL connection
const db = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "exp2"
});

db.connect((err) => {
    if (err) throw err;
    console.log("Connected to MySQL");
});

const server = http.createServer((req, res) => {

    if (req.method === "GET" && req.url === "/") {
        fs.readFile("login.html", (err, data) => {
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(data);
        });
```

```
}

else if (req.method === "GET" && req.url === "/styles.css") {
    fs.readFile("styles.css", (err, data) => {
        res.writeHead(200, { "Content-Type": "text/css" });
        res.end(data);
    });
}

else if (req.method === "POST" && req.url === "/login") {
    let body = "";

    req.on("data", chunk => {
        body += chunk.toString();
    });

    req.on("end", () => {
        const { username, password } = querystring.parse(body);

        const sql = "SELECT * FROM users WHERE username = ? AND password = ? LIMIT 1";

        db.query(sql, [username, password], (err, results) => {
            if (err) throw err;

            res.writeHead(200, { "Content-Type": "text/html" });

            if (results.length > 0) {
                res.end(`<h2 style="color:green;text-align:center;">Login Successful! Welcome
${username}</h2>`);
            } else {
                res.end(`<h2 style="color:red;text-align:center;">Invalid Username or Password</h2>`);
            }
        });
    });
  }

  else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("404 Not Found");
  }
});

server.listen(3000, () => {
  console.log("Server running at http://localhost:3000");
});
```

---

index.html :

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Simple Login</title>
  <link rel="stylesheet" href="styles.css">
```

```
</head>
<body>
<div class="container">
    <h2>Login</h2>
    <form method="POST" action="/login">
        <input type="text" name="username" placeholder="Enter Username" required>
        <input type="password" name="password" placeholder="Enter Password" required>
        <button type="submit">Login</button>
    </form>

    <p id="message"></p>
</div>
</body>
</html>
```

styles.css :

```
body {
    font-family: Arial, sans-serif;
    background: #f4f4f4;
    display: flex;
    height: 100vh;
    align-items: center;
    justify-content: center;
}
.container {
    background: white;
    padding: 25px;
    border-radius: 10px;
    width: 300px;
    box-shadow: 0 0 10px rgba(0,0,0,0.1);
}
h2 {
    text-align: center;
}
input {
    width: 100%;
    padding: 10px;
    margin: 8px 0;
    border-radius: 5px;
    border: 1px solid #aaa;
}
button {
    width: 100%;
    padding: 10px;
    background: #007bff;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
button:hover {
    background: #0056b3;
}
```

EXPERIMENT – 3

Write a Node JS program to perform read, write and other operations on a file.

AIM: To write a Node.js program that performs **write**, **read**, **append**, **rename**, and **delete** operations on a file using the built-in `fs` module.

OBJECTIVE:

- To understand the use of Node.js **File System (fs)** module.
- To create, read, update, rename, and delete files programmatically.
- To learn asynchronous callback-based file handling.

THEORY:

Node.js provides the **fs (File System)** module to work with files.
It allows both **synchronous** and **asynchronous** operations such as:

- `writeFile()` → Create/write file
- `readFile()` → Read file
- `appendFile()` → Add data to existing file
- `rename()` → Change file name
- `unlink()` → Delete a file

In this experiment, we use **asynchronous** functions because they do not block the main thread.

PROGRAM:
file_operations.js

```
const fs = require("fs");

// 1. Write data to a file
fs.writeFile("sample.txt", "This is the initial content.\n", (err) => {
    if (err) throw err;
    console.log("File created and data written!");

    // 2. Read the file
    fs.readFile("sample.txt", "utf8", (err, data) => {
        if (err) throw err;
        console.log("File Content:");
        console.log(data);

        // 3. Append data to the file
        fs.appendFile("sample.txt", "This line is appended.\n", (err) => {
            if (err) throw err;
            console.log("Data appended to file!");

            // 4. Rename the file
            fs.rename("sample.txt", "new_sample.txt", (err) => {
                if (err) throw err;
                console.log("File renamed to new_sample.txt");
```

```
            // 5. Delete the file
            fs.unlink("new_sample.txt", (err) => {
                if (err) throw err;
                console.log("File deleted successfully!");
            });
        });
    });
});
```

Run Procedure (Output):

```
C:\> md exp3
C:\exp3>node file_operations.js
File created and data written!
File Content:
This is the initial content.

Data appended to file!
File renamed to new_sample.txt
File deleted successfully!
```

EXPERIMENT – 4

Write a Node JS program to read form data from query string and generate response using NodeJS

AIM

To write a Node.js program that reads form data sent through a **query string** and generates an appropriate response.

OBJECTIVE

- To understand how Node.js handles HTTP GET requests.
- To extract parameters from a URL query string.
- To generate dynamic output based on user input.

THEORY

In HTTP GET requests, form data is sent in the URL after a `?` symbol in key–value pairs.
Example:

```
http://localhost:3000/?name=John&age=20
```

Node.js provides the built-in **url** module to parse query strings:

- `url.parse(req.url, true).query` gives an object with form data.

PROGRAM:

server.js

```
const http = require("http");
const url = require("url");

const server = http.createServer((req, res) => {

    // Parse URL and get query string data
    const q = url.parse(req.url, true).query;

    // If no data, show form
    if (!q.name && !q.age) {
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(`
                    <center><div>
            <h2>Enter Your Details</h2>
            <form method="GET" action="/">
                <input type="text" name="name" placeholder="Enter Name" required><br><br>
                <input type="number" name="age" placeholder="Enter Age" required><br><br>
                <button type="submit">Submit</button>
            </form></div></center>
        `);
    }
```

```
    else {
        // Data available → generate response
        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(`
            <h2>Form Submitted Successfully!</h2>
            <p><b>Name:</b> ${q.name}</p>
            <p><b>Age:</b> ${q.age}</p>
        `);
    }
});

// Start server
server.listen(3000, () => {
    console.log("Server running at http://localhost:3000");
});
```

OUTPUT

Case 1: Initial Page

Displays form:

After submitting

Example:
URL → `http://localhost:3000/?name=mahesh&age=25`

EXPERIMENT – 5

Create a food delivery website where users can order food from a particular restaurant listed in the website for handling http requests and responses using NodeJS.

AIM:

To create a simple food delivery website using Node.js where restaurant names, menu items, and prices are stored in a JSON file**, and to handle HTTP requests and responses.**

OBJECTIVE

- To understand HTTP request and response handling in Node.js
- To create a basic web server using Node.js
- To display restaurant and food items
- To process user order input

THEORY

Node.js can read external files using the **fs module**: In this experiment:

- Restaurant and menu details are stored in a **JSON file**
- Node.js reads the JSON file and displays restaurants and menus
- User selects food items
- Server calculates the total bill and sends response
- `styles.css` is used for styling the pages

PROCEDURE:

- Create files: `server.js`, `data.json`, `styles.css`
- Run:

C:\exp5> node server.js

PROGRAM:

data.json:

```
{
 "restaurants": [
   {
    "id": "spicyhub",
    "name": "Spicy Hub",
    "menu": {
     "Biryani": 200,
     "Paneer": 150,
     "Roti": 20
    }
```

```
    },
    {
      "id": "italiandelight",
      "name": "Italian Delight",
      "menu": {
        "Pizza": 250,
        "Pasta": 180,
        "Garlic Bread": 120
      }
    }
  ]
}
```

server.js

```
const http = require("http");
const fs = require("fs");
const querystring = require("querystring");

// Read JSON data
const data = JSON.parse(fs.readFileSync("data.json", "utf8"));

const server = http.createServer((req, res) => {

    // Serve CSS
    if (req.url === "/styles.css") {
        fs.readFile("styles.css", (err, css) => {
            res.writeHead(200, { "Content-Type": "text/css" });
            res.end(css);
        });
    }

    // Home Page - Restaurant List
    else if (req.url === "/" && req.method === "GET") {
        let restHtml = "";
        data.restaurants.forEach(r => {
            restHtml += `<a href="/menu?rest=${r.id}" class="btn">${r.name}</a><br><br>`;
        });

        res.writeHead(200, { "Content-Type": "text/html" });
        res.end(`
            <html>
            <head>
                <title>Food Delivery</title>
                <link rel="stylesheet" href="styles.css">
            </head>
            <body>
                <div class="container">
                    <h2>Food Delivery Website</h2>
                    <h3>Select Restaurant</h3>
                    ${restHtml}
                </div>
            </body>
            </html>
        `);
```

```
}

// Menu Page
else if (req.url.startsWith("/menu") && req.method === "GET") {
    const query = new URL(req.url, "http://localhost").searchParams;
    const restId = query.get("rest");
    const restaurant = data.restaurants.find(r => r.id === restId);

    if (!restaurant) {
        res.writeHead(404, { "Content-Type": "text/html" });
        res.end("<h3>Restaurant not found</h3>");
        return;
    }

    let menuHtml = "";
    for (let item in restaurant.menu) {
        menuHtml += `
            <input type="checkbox" name="food" value="${item}">
            ${item} (Rs. ${restaurant.menu[item]})<br>
        `;
}

res.writeHead(200, { "Content-Type": "text/html" });
res.end(`
    <html>
    <head>
        <title>${restaurant.name}</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <div class="container">
            <h2>${restaurant.name} - Menu</h2>
            <form method="POST" action="/order?rest=${restaurant.id}">
                ${menuHtml}<br>
                <button type="submit" class="btn">Place Order</button>
            </form>
            <br>
            <a href="/" class="link">Back</a>
        </div>
    </body>
    </html>
`);
}

// Order Confirmation Page
else if (req.url.startsWith("/order") && req.method === "POST") {
    const query = new URL(req.url, "http://localhost").searchParams;
    const restId = query.get("rest");
    const restaurant = data.restaurants.find(r => r.id === restId);

    let body = "";
    req.on("data", chunk => body += chunk.toString());

    req.on("end", () => {
```

```
let formData = querystring.parse(body);
let items = formData.food || [];

if (!Array.isArray(items)) items = [items];

let total = 0;
items.forEach(item => {
    total += restaurant.menu[item];
});

res.writeHead(200, { "Content-Type": "text/html" });
res.end(`
    <html>
    <head>
        <title>Order Confirmation</title>
        <link rel="stylesheet" href="styles.css">
    </head>
    <body>
        <div class="container">
            <h2>Order Confirmed</h2>
            <p><b>Restaurant:</b> ${restaurant.name}</p>
            <p><b>Items Ordered:</b> ${items.join(", ")}</p>
            <p><b>Total Amount to Pay:</b> ${total}</p>
            <p class="success">Thank you for your order!</p>
            <a href="/" class="btn">Home</a>
        </div>
    </body>
    </html>
    `);
  });
}

// Invalid URL
else {
  res.writeHead(404, { "Content-Type": "text/html" });
  res.end("<h3>404 - Page Not Found</h3>");
}
});

// Start Server
server.listen(3000, () => {
  console.log("Server running at http://localhost:3000");
});
```

EXPERIMENT – 6

Title: Implement a program with basic commands on databases and collections using MongoDB.

AIM: To implement a Node.js program that performs basic MongoDB operations such as creating a database, creating a collection, inserting, reading, updating, and deleting documents.

Objectives

- To understand MongoDB databases and collections
- To perform CRUD operations in MongoDB
- To connect Node.js with MongoDB
- To execute basic MongoDB commands programmatically

MongoDB is a NoSQL, document-oriented database that stores data in JSON-like documents.
In MongoDB:

- A **database** contains collections
- A **collection** contains documents
- Documents are stored in **key–value format**

CRUD operations include:

- **Create** → `insertOne()`
- **Read** → `find()`
- **Update** → `updateOne()`
- **Delete** → `deleteOne()`

Node.js connects to MongoDB using the official **mongodb driver**.

Procedure

1. Start MongoDB server
2. Create a project folder
3. Initialize Node.js project
4. Install MongoDB driver
5. Write program to perform CRUD operations
6. Execute the program

Program:

File name: mongo_basic.js

```
// Import MongoClient from mongodb package
const { MongoClient } = require("mongodb");

// MongoDB connection URL
const url = "mongodb://localhost:27017";

// Create MongoClient instance
const client = new MongoClient(url);

// Database and collection names
const dbName = "collegeDB";
const collectionName = "students";

// Async function to perform MongoDB operations
async function run() {
 try {
  // Connect to MongoDB
  await client.connect();
  console.log("Connected to MongoDB");

  // Select database
  const db = client.db(dbName);

  // Select collection
  const collection = db.collection(collectionName);

  // INSERT document
  await collection.insertOne({
   rollno: 101,
   name: "Ravi",
   branch: "CSE",
   marks: 85
  });
  console.log("Document Inserted");

  // READ documents
  const students = await collection.find({}).toArray();
  console.log("Documents in Collection:");
  console.log(students);

  // UPDATE document
  await collection.updateOne(
   { rollno: 101 },
   { $set: { marks: 90 } }
  );
  console.log("Document Updated");

  // DELETE document
  await collection.deleteOne({ rollno: 101 });
  console.log("Document Deleted");

 } catch (error) {
  console.error(error);
```

```
  } finally {
    // Close MongoDB connection
    await client.close();
    console.log("Connection Closed");
  }
}

// Call function
Run();
```

Execution:

Output

```
Connected to MongoDB
Document Inserted
Documents in Collection:
[ { rollno: 101, name: 'Ravi', branch: 'CSE', marks: 85 } ]
Document Updated
Document Deleted
Connection Closed
```

EXPERIMENT – 7

Title : Implement CRUD operations on the given dataset using MongoDB.

Aim

To implement Create, Read, Update, and Delete (CRUD) operations on a given dataset using MongoDB and Node.js.

Objectives

- To understand CRUD operations in MongoDB
- To store and manipulate data in collections
- To connect MongoDB with Node.js
- To perform insert, read, update, and delete operations

Theory

MongoDB is a NoSQL database that stores data in **document format (JSON-like)**.

CRUD operations are the basic database operations:

- **Create** → `insertOne()` / `insertMany()`
- **Read** → `find()`
- **Update** → `updateOne()`
- **Delete** → `deleteOne()`

Node.js interacts with MongoDB using the official **mongodb driver**.

Program :

File Name: `crud_operations.js`

```
// Import MongoClient
const { MongoClient } = require("mongodb");
// MongoDB URL
const url = "mongodb://localhost:27017";
// Create client
const client = new MongoClient(url);
// Database and collection names
const dbName = "collegeDB";
const collectionName = "students";

async function run() {
  try {
```

```
// Connect to MongoDB
await client.connect();
console.log("Connected to MongoDB");
// Access database and collection
const db = client.db(dbName);
const collection = db.collection(collectionName);
// ---------------- CREATE ----------------
await collection.insertMany([
  { rollno: 101, name: "Ravi", branch: "CSE", marks: 85 },
  { rollno: 102, name: "Sita", branch: "ECE", marks: 78 },
  { rollno: 103, name: "Kiran", branch: "CSE", marks: 92 }
]);
console.log("Documents Inserted");
// ---------------- READ ----------------
const students = await collection.find({}).toArray();
console.log("All Students:");
console.log(students);
// ---------------- UPDATE ----------------
await collection.updateOne(
  { rollno: 102 },
  { $set: { marks: 82 } }
  );
  console.log("Document Updated");
  // ---------------- DELETE ----------------
  await collection.deleteOne({ rollno: 101 });
  console.log("Document Deleted");
} catch (err) {
  console.error(err);
} finally {
  // Close connection
```

```
    await client.close();
    console.log("Connection Closed");
  }
}
// Execute function
run();
```

Output :

Connected to MongoDB

Documents Inserted

All Students:

[

{ rollno: 101, name: 'Ravi', branch: 'CSE', marks: 85 },

{ rollno: 102, name: 'Sita', branch: 'ECE', marks: 78 },

{ rollno: 103, name: 'Kiran', branch: 'CSE', marks: 92 }

]

Document Updated

Document Deleted

Connection Closed.

EXPERIMENT – 8

Title : Perform Count, Limit, Sort, and Skip operations on the given collections using MongoDB.

Aim :

To perform Count, Limit, Sort, and Skip operations on a MongoDB collection using Node.js.

Objectives

- To understand advanced read operations in MongoDB
- To retrieve limited and sorted records
- To implement pagination using skip and limit
- To connect Node.js with MongoDB

Theory

MongoDB provides several query methods to control the number and order of documents returned.

- **Count** → Counts number of documents
- **Limit** → Restricts number of documents returned
- **Sort** → Arranges documents in ascending or descending order
- **Skip** → Skips specified number of documents

These operations are commonly used in pagination and report generation.

Dataset Used (students collection)

rollno	name	branch	marks
101	Ravi	CSE	85
102	Sita	ECE	78
103	Kiran	CSE	92
104	Manoj	IT	88
105	Anita	CSE	90

Program

File Name: `count_limit_sort_skip.js`

```
// Import MongoClient

const { MongoClient } = require("mongodb");

// MongoDB URL

const url = "mongodb://localhost:27017";

// Create MongoClient instance

const client = new MongoClient(url);

// Database and collection names

const dbName = "collegeDB";

const collectionName = "students";

async function run() {

  try {

    // Connect to MongoDB

    await client.connect();

    console.log("Connected to MongoDB");

    const db = client.db(dbName);

    const collection = db.collection(collectionName);

    // --------------- COUNT --------------

    const count = await collection.countDocuments();

    console.log("Total Number of Students:", count);

    // --------------- SORT ---------------

    const sortedStudents = await collection

      .find({})
```

```
  .sort({ marks: -1 })   // Descending order
  .toArray();
console.log("Students Sorted by Marks (Descending):");
console.log(sortedStudents);
// --------------- LIMIT ---------------
const limitedStudents = await collection
  .find({})
  .limit(3)
  .toArray();
console.log("First 3 Students:");
console.log(limitedStudents);
// --------------- SKIP + LIMIT ---------------
const skippedStudents = await collection
  .find({})
  .skip(2)
  .limit(2)
  .toArray();
  console.log("Students After Skipping 2 Records:");
  console.log(skippedStudents);
} catch (error) {
  console.error(error);
} finally {
  // Close connection
  await client.close();
```

```
    console.log("Connection Closed");
  }
}
// Execute function
run();
```

Output :

Connected to MongoDB

Total Number of Students: 5

Students Sorted by Marks (Descending):

[Kiran, Anita, Manoj, Ravi, Sita]

First 3 Students:

[Ravi, Sita, Kiran]

Students After Skipping 2 Records:

[Kiran, Manoj]

Connection Closed

EXPERIMENT – 9

Title: Develop an AngularJS form to apply CSS and Events.

AIM: To develop an AngularJS form and apply CSS styling and event handling using AngularJS directives.

Objectives

- To design a form using AngularJS
- To apply CSS styles dynamically
- To handle user events like button click
- To understand AngularJS directives.

Theory

AngularJS is a JavaScript framework used to create dynamic web applications.
It supports:

- **Two-way data binding**
- **Directives** such as `ng-app`, `ng-model`, `ng-click`
- **Event handling** without writing JavaScript separately

CSS is used to improve the appearance of HTML elements, while AngularJS events respond to user actions.

Procedure

1. Create an HTML file
2. Include AngularJS library
3. Define AngularJS application and controller
4. Design a form with input fields
5. Apply CSS styles
6. Handle events using AngularJS directives

Program

File Name: `index.html`

```
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
 <meta charset="UTF-8">
 <title>AngularJS Form with CSS and Events</title>

 <!-- AngularJS Library -->
 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

 <!-- CSS Styling -->
 <style>
  body {
   font-family: Arial;
```

```
  background-color: #f2f2f2;
}
.container {
  width: 350px;
  margin: auto;
  margin-top: 80px;
  padding: 20px;
  background: white;
  border-radius: 10px;
  box-shadow: 0px 0px 10px gray;
}
h2 {
  text-align: center;
  color: #007bff;
}
input {
  width: 100%;
  padding: 10px;
  margin: 8px 0;
  border-radius: 5px;
  border: 1px solid #aaa;
 }
 button {
  width: 100%;
  padding: 10px;
  background-color: green;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
 }
 button:hover {
  background-color: darkgreen;
 }
 .result {
  margin-top: 15px;
  color: blue;
  text-align: center;
 }
 </style>
</head>

<body ng-controller="myCtrl">

 <div class="container">
  <h2>User Form</h2>

  <label>Name</label>
  <input type="text" ng-model="name" placeholder="Enter Name">

  <label>Email</label>
  <input type="email" ng-model="email" placeholder="Enter Email">

  <button ng-click="submit()">Submit</button>

  <div class="result" ng-if="submitted">
   <p>Name: {{name}}</p>
```

```
    <p>Email: {{email}}</p>
   </div>
  </div>

  <script>
   var app = angular.module("myApp", []);
   app.controller("myCtrl", function($scope) {
    $scope.submitted = false;

    $scope.submit = function() {
     $scope.submitted = true;
    };
   });
  </script>

</body>
</html>
```

Execution:

Output

- Displays a styled form
- Accepts user input
- On button click, entered details are displayed dynamically

EXPERIMENT – 10

Title: Develop a Job Registration form and validate it using AngularJS.

AIM: To develop a Job Registration form using AngularJS and perform client-side form validation.

Objectives

- To design a job registration form
- To apply AngularJS form validation
- To understand AngularJS validation directives
- To prevent submission of invalid data

Theory

AngularJS provides built-in form validation using directives such as:

- `ng-model`
- `ng-required`
- `ng-pattern`
- `$valid`, `$invalid`, `$touched`

AngularJS automatically tracks form and input states and displays error messages dynamically without page reload.

Procedure

1. Create an HTML file
2. Include AngularJS library
3. Define AngularJS module and controller
4. Design Job Registration form
5. Apply validation rules
6. Display error messages
7. Run the application in browser

Program

File Name: `job_registration.html`

```
<!DOCTYPE html>
<html lang="en" ng-app="jobApp">
<head>
  <meta charset="UTF-8">
  <title>Job Registration Form</title>
```

```
<!-- AngularJS Library -->
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<style>
  body {
    font-family: Arial;
    background-color: #f2f2f2;
  }
  .container {
    width: 400px;
    margin: auto;
    margin-top: 60px;
    background: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 0px 10px gray;
}
h2 {
  text-align: center;
  color: #007bff;
}
input, select {
  width: 100%;
  padding: 8px;
  margin: 6px 0;
}
button {
  width: 100%;
  padding: 10px;
  background: green;
    color: white;
    border: none;
    border-radius: 5px;
  }
  .error {
    color: red;
    font-size: 13px;
  }
  .success {
    color: green;
    text-align: center;
    margin-top: 10px;
  }
</style>
```

```
</head>

<body ng-controller="jobCtrl">

 <div class="container">
  <h2>Job Registration</h2>
  <form name="jobForm" novalidate>
   <!-- Name -->
   <label>Full Name</label>
   <input type="text" name="name"
        ng-model="user.name"
        ng-required="true">
   <span class="error"
       ng-show="jobForm.name.$touched && jobForm.name.$invalid">
     Name is required
</span>

<!-- Email -->
<label>Email</label>
<input type="email" name="email"
      ng-model="user.email"
      ng-required="true">
<span class="error"
     ng-show="jobForm.email.$touched && jobForm.email.$invalid">
     Enter valid email
</span>
<!-- Mobile -->
<label>Mobile Number</label>
<input type="text" name="mobile"
      ng-model="user.mobile"
      ng-pattern="/^[0-9]{10}$/"
      ng-required="true">
<span class="error"
     ng-show="jobForm.mobile.$touched && jobForm.mobile.$invalid">
     Enter 10 digit mobile number
</span>
<!-- Qualification -->
<label>Qualification</label>
<select name="qualification"
       ng-model="user.qualification"
       ng-required="true">
  <option value="">Select</option>
  <option>B.Tech</option>
  <option>M.Tech</option>
  <option>MCA</option>
```

```
    <option>B.Sc</option>
  </select>
  <span class="error"
      ng-show="jobForm.qualification.$touched && jobForm.qualification.$invalid">
      Select qualification
  </span>

  <!-- Submit -->
  <button type="submit"
        ng-click="register()"
        ng-disabled="jobForm.$invalid">
        Register
  </button>

 </form>

  <div class="success" ng-if="success">
    Registration Successful!
  </div>

</div>

<script>
  var app = angular.module("jobApp", []);
  app.controller("jobCtrl", function($scope) {
    $scope.success = false;

    $scope.register = function() {
      if ($scope.jobForm.$valid) {
        $scope.success = true;
      }
    };
   });
  </script>

</body>
</html>
```

Output

- Displays Job Registration Form
- Shows error messages for invalid inputs
- Submit button enabled only when form is valid
- Displays **Registration Successful** message on valid submission

EXPERIMENT – 11

Title: Write an AngularJS application to access JSON file data of an employee from a server using `$http` service.

AIM: To develop an AngularJS application that retrieves employee data from a JSON file using the `$http` service and displays it on a web page.

Objectives

- To understand `$http` service in AngularJS
- To fetch data from a JSON file (server)
- To display dynamic data using AngularJS
- To understand promises (`then`, `catch`)

Theory:

AngularJS provides the `$http` service to communicate with servers using HTTP methods like **GET, POST, PUT, DELETE**.

In this experiment:

- Employee details are stored in a **JSON file**
- `$http.get()` fetches data from the server
- Retrieved data is bound to the view using AngularJS expressions

Procedure

1. Create a JSON file with employee data
2. Create an HTML file
3. Include AngularJS library
4. Define AngularJS module and controller
5. Use `$http.get()` to fetch JSON data
6. Display employee details in a table

Program

File 1: `employees.json`

```
[
  {
    "empId": 101,
    "name": "Ravi",
    "designation": "Software Engineer",
    "salary": 45000
  },
```

```
  {
    "empId": 102,
    "name": "Sita",
    "designation": "Tester",
    "salary": 38000
  },
  {
    "empId": 103,
    "name": "Kiran",
    "designation": "Project Manager",
    "salary": 60000
  }
]
```

File 2: `index.html`

```
<!DOCTYPE html>
<html lang="en" ng-app="empApp">
<head>
  <meta charset="UTF-8">
  <title>Employee Details</title>

  <!-- AngularJS Library -->
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

  <style>
    body {
      font-family: Arial;
      background-color: #f2f2f2;
    }
    .container {
      width: 600px;
      margin: auto;
      margin-top: 60px;
      background: white;
      padding: 20px;
      border-radius: 10px;
      box-shadow: 0px 0px 10px gray;
    }
    h2 {
      text-align: center;
      color: #007bff;
    }
    table {
      width: 100%;
```

```
    border-collapse: collapse;
    margin-top: 15px;
   }
   th, td {
    padding: 10px;
    border: 1px solid #aaa;
    text-align: center;
   }
   th {
    background-color: #007bff;
    color: white;
   }
  </style>
</head>

<body ng-controller="empCtrl">

 <div class="container">
  <h2>Employee Details</h2>

  <table>
   <tr>
    <th>Employee ID</th>
    <th>Name</th>
    <th>Designation</th>
    <th>Salary</th>
   </tr>
   <tr ng-repeat="emp in employees">
    <td>{{emp.empId}}</td>
    <td>{{emp.name}}</td>
    <td>{{emp.designation}}</td>
    <td>{{emp.salary}}</td>
   </tr>
  </table>
</div>

<script>
 var app = angular.module("empApp", []);

 app.controller("empCtrl", function($scope, $http) {

  // Fetch JSON data using $http service
  $http.get("employees.json")
   .then(function(response) {
    $scope.employees = response.data;
```

```
      })
      .catch(function(error) {
        console.error("Error fetching data", error);
      });

  });
 </script>

</body>
</html>
```

Output

- Employee details are fetched from `employees.json`
- Data is displayed dynamically in a table
- Page updates without reload

EXPERIMENT – 12

Title: Develop a web application to manage student information using Express and AngularJS.

AIM: To develop a web application using Express (Node.js) and AngularJS to manage student information such as adding and viewing student details.

Objectives

- To understand Express framework
- To create REST API using Express
- To fetch data using AngularJS `$http` service
- To manage student information dynamically

Theory

Express.js is a lightweight Node.js framework used to create web servers and REST APIs.

AngularJS is a front-end framework used to build dynamic single-page applications.

In this application:

- Express acts as **backend server**
- AngularJS acts as **frontend**
- Student data is exchanged using **JSON format**

Procedure

1. Create project folder
2. Initialize Node.js project
3. Install Express
4. Create server using Express
5. Create AngularJS frontend
6. Perform add and view student operations

Project Structure

```
exp12/
 ├── server.js
 └── public/
     └── index.html
```

Program

File 1: `server.js` (Express Backend)

```
const express = require("express");
```

```
const app = express();
const port = 3000;

app.use(express.json());
app.use(express.static("public"));

// In-memory student data
let students = [
  { rollno: 101, name: "Ravi", branch: "CSE" },
  { rollno: 102, name: "Sita", branch: "ECE" }
];

// READ – Get all students
app.get("/students", (req, res) => {
  res.json(students);
});

// CREATE – Add new student
app.post("/students", (req, res) => {
  students.push(req.body);
  res.json({ message: "Student Added" });
});

// UPDATE – Edit student
app.put("/students/:rollno", (req, res) => {
  const roll = parseInt(req.params.rollno);
  const index = students.findIndex(s => s.rollno === roll);

  if (index !== -1) {
    students[index] = req.body;
    res.json({ message: "Student Updated" });
  } else {
    res.status(404).json({ message: "Student Not Found" });
  }
});

// DELETE – Remove student
app.delete("/students/:rollno", (req, res) => {
  const roll = parseInt(req.params.rollno);
  students = students.filter(s => s.rollno !== roll);
  res.json({ message: "Student Deleted" });
});

// Start server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

File 2: `public/index.html` (AngularJS Frontend):

```
<!DOCTYPE html>
<html ng-app="studentApp">
```

```
<head>
  <title>Student Management</title>

  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

  <style>
    body { font-family: Arial; background: #f2f2f2; }
    .container {
      width: 600px;
      margin: auto;
      background: white;
      padding: 20px;
      margin-top: 40px;
      border-radius: 10px;
    }
    input {
      width: 100%;
      padding: 8px;
      margin: 6px 0;
  }
  button {
    padding: 8px;
    margin: 5px;
    border: none;
    cursor: pointer;
  }
  .add { background: green; color: white; width: 100%; }
  .edit { background: orange; color: white; }
  .delete { background: red; color: white; }
  table {
    width: 100%;
    margin-top: 15px;
    border-collapse: collapse;
  }
  th, td {
    border: 1px solid #aaa;
      padding: 8px;
      text-align: center;
    }
    th {
      background: #007bff;
      color: white;
    }
  </style>
</head>

<body ng-controller="studentCtrl">

<div class="container">
  <h2 align="center">Student Management System</h2>

  <input type="number" ng-model="student.rollno" placeholder="Roll No">
```

```
<input type="text" ng-model="student.name" placeholder="Name">
<input type="text" ng-model="student.branch" placeholder="Branch">

<button class="add" ng-click="saveStudent()">
 {{ editMode ? 'Update Student' : 'Add Student' }}
</button>

<table>
 <tr>
  <th>Roll No</th>
  <th>Name</th>
  <th>Branch</th>
  <th>Actions</th>
 </tr>
 <tr ng-repeat="s in students">
  <td>{{s.rollno}}</td>
  <td>{{s.name}}</td>
  <td>{{s.branch}}</td>
  <td>
   <button class="edit" ng-click="editStudent(s)">Edit</button>
   <button class="delete" ng-click="deleteStudent(s.rollno)">Delete</button>
  </td>
 </tr>
</table>
</div>

<script>
 var app = angular.module("studentApp", []);

 app.controller("studentCtrl", function($scope, $http) {

  $scope.editMode = false;

  // Load students
  function loadStudents() {
   $http.get("/students").then(function(res) {
    $scope.students = res.data;
   });
  }
  loadStudents();

  // Add or Update student
  $scope.saveStudent = function() {
   if ($scope.editMode) {
    $http.put("/students/" + $scope.student.rollno, $scope.student)
     .then(function() {
      loadStudents();
      $scope.student = {};
      $scope.editMode = false;
     });
   } else {
    $http.post("/students", $scope.student)
```

```
      .then(function() {
        loadStudents();
        $scope.student = {};
      });
  }
};

// Edit student
$scope.editStudent = function(s) {
  $scope.student = angular.copy(s);
  $scope.editMode = true;
};

// Delete student
$scope.deleteStudent = function(rollno) {
  $http.delete("/students/" + rollno)
    .then(function() {
      loadStudents();
    });
  };

});
</script>

</body>
</html>
```

How to Run:

```
cd exp12
npm init -y
npm install express
node server.js
```

Open browser: http://localhost:3000

Student Management web application with Add, View, Edit, and Delete operations was successfully developed using Express and AngularJS.

EXPERIMENT – 13

Title: Write a program to create a simple calculator Application using React JS.

AIM:To develop a simple calculator application using React JS to perform basic arithmetic operations.

Objectives

- To understand React JS components
- To use `useState` hook
- To perform arithmetic operations
- To handle events in React JS

Theory

React JS is a JavaScript library used to build user interfaces using **components**.
A calculator application demonstrates:

- Component-based architecture
- State management using `useState`
- Event handling using `onClick`

Procedure

1. Create React application
2. Modify `App.js` file
3. Add calculator UI
4. Implement arithmetic logic
5. Run the application

Program

Step 1: Create React App

npx create-react-app calculator-app
cd calculator-app
npm start

Step 2: `App.js`

```
import React, { useState } from "react";
import "./App.css";

function App() {
  const [num1, setNum1] = useState("");
  const [num2, setNum2] = useState("");
  const [result, setResult] = useState("");
  return (
    <div className="container">
      <h2>Simple Calculator</h2>
      <input
        type="number"
        placeholder="Enter Number 1"
        value={num1}
        onChange={(e) => setNum1(e.target.value)}
      />
      <input
        type="number"
        placeholder="Enter Number 2"
```

```
      value={num2}
      onChange={(e) => setNum2(e.target.value)}
    />
    <div className="buttons">
      <button onClick={() => setResult(Number(num1) + Number(num2))}>+</button>
      <button onClick={() => setResult(Number(num1) - Number(num2))}>-</button>
      <button onClick={() => setResult(Number(num1) * Number(num2))}>*</button>
      <button onClick={() => setResult(Number(num1) / Number(num2))}>/</button>
    </div>

    <h3>Result: {result}</h3>
   </div>
  );
}
export default App;
```

Step 3: **App.css**

```
body {
  background: #f2f2f2;
  font-family: Arial;
}

.container {
  width: 300px;
  margin: auto;
  margin-top: 80px;
  background: white;
  padding: 20px;
  border-radius: 10px;
  text-align: center;
  box-shadow: 0 0 10px gray;
}

input {
  width: 100%;
  padding: 8px;
  margin: 8px 0;
}

.buttons button {
  width: 50px;
  height: 40px;
  margin: 5px;
  font-size: 18px;
  cursor: pointer;
}
```

Output

- Displays calculator interface
- Performs addition, subtraction, multiplication, and division
- Result updates dynamically without page reload

EXPERIMENT – 14

Title: Write a program to create a voting application using React JS

AIM:To develop a simple voting application using React JS that allows users to vote for candidates and displays vote counts dynamically.

Objectives

- To understand React JS functional components
- To use `useState` hook for state management
- To handle button click events
- To update UI dynamically without page reload

Theory

React JS is a component-based JavaScript library used to build interactive user interfaces.
In a voting application:

- Each candidate has a vote count
- Clicking a button updates the state
- React automatically re-renders the UI

Program :

Step 1: Create React App

npx create-react-app voting-app

cd voting-app

npm start

Step 2: `App.js`

```
import React, { useState } from "react";
import "./App.css";
function App() {
  const [votesA, setVotesA] = useState(0);
  const [votesB, setVotesB] = useState(0);
  const [votesC, setVotesC] = useState(0);

  return (
    <div className="container">
      <h2>Voting Application</h2>
      <div className="card">
        <h3>Candidate A</h3>
        <p>Votes: {votesA}</p>
        <button onClick={() => setVotesA(votesA + 1)}>Vote</button>
      </div>
      <div className="card">
        <h3>Candidate B</h3>
        <p>Votes: {votesB}</p>
        <button onClick={() => setVotesB(votesB + 1)}>Vote</button>
```

```
      </div>
      <div className="card">
        <h3>Candidate C</h3>
        <p>Votes: {votesC}</p>
        <button onClick={() => setVotesC(votesC + 1)}>Vote</button>
      </div>
    </div>
  );
}
export default App;
```

Step 3: App.css

```
body {
  background-color: #f2f2f2;
  font-family: Arial;
}
.container {
  width: 400px;
  margin: auto;
  margin-top: 60px;
  background: white;
  padding: 20px;
  border-radius: 10px;
  text-align: center;
  box-shadow: 0 0 10px gray;
}
.card {
  border: 1px solid #ccc;
  margin: 10px 0;
  padding: 15px;
  border-radius: 8px;
}
button {
  padding: 8px 15px;
  background-color: green;
  color: white;
  border: none;
  cursor: pointer;
  border-radius: 5px;
}
button:hover {
  background-color: darkgreen;
}
```

Output

- Displays list of candidates
- Allows user to vote
- Vote count increases dynamically